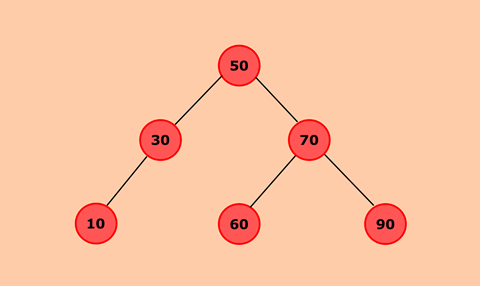
**Binary Search Tree**



In **Binary Search Tree**, all nodes which are present to the left of root will be less than root node and nodes which are present to the right will be greater than the root node.

### **Insertion:**

* If the value of the new node is less than the root node then, it will be inserted to the left subtree.
* If the value of the new node is greater than root node then, it will be inserted to the right subtree.

### **Deletion:**

* If the node to be deleted is a leaf node then, parent of that node will point to null. For eg. If we delete 90, then parent node 70 will point to null.
* If the node to be deleted has one child node, then child node will become a child node of the parent node. For eg. If we delete 30, then node 10 which was left child of 30 will become left child of 50.
* If the node to be deleted has two children then, we find the node(minNode) with minimum value from the right subtree of that current node. The current node will be replaced by its successor(minNode).

#### **Insert An Element In BST**

An element is always inserted as a leaf node in BST.

**Given below are the steps for inserting an element.**

1. Start from the root.
2. Compare the element to be inserted with the root node. If it is less than root, then traverse the left subtree or traverse the right subtree.
3. Traverse the subtree till the end of the desired subtree. Insert the node in the appropriate subtree as a leaf node.

#### **Search Operation In BST**

To search if an element is present in the BST, we again start from the root and then traverse the left or right subtree depending on whether the element to be searched is less than or greater than the root.

**Enlisted below are the steps that we have to follow.**

1. Compare the element to be searched with the root node.
2. If the key (element to be searched) = root, return root node.
3. Else if key < root, traverse the left subtree.
4. Else traverse right subtree.
5. Repetitively compare subtree elements until the key is found or the end of the tree is reached.

#### **Remove Element From The BST**

When we delete a node from the BST, then there are three possibilities as discussed below:

**Node Is A Leaf Node**

If a node to be deleted is a leaf node, then we can directly delete this node as it has no child nodes.

**Node Has Only One Child**

When we need to delete the node that has one child, then we copy the value of the child in the node and then delete the child.

**Node Has Two Children**

When a node to be deleted has two children, then we replace the node with the inorder (left-root-right) successor of the node or simply said the minimum node in the right subtree if the right subtree of the node is not empty. We replace the node with this minimum node and delete the node.

**Implementation:**

**public** **class** BinarySearchTree

 {

    //Represent a node of binary tree

**public** **static** **class** Node

{

**int** data;

        Node left;

        Node right;

**public** Node(**int** data)

{

            //Assign data to the new node, set left and right children to null

**this**.data = data;

**this**.left = **null**;

**this**.right = **null**;

        }

      }

      //Represent the root of binary tree

**public** Node root;

**public** BinarySearchTree()

{

          root = **null**;

      }

      //insert() will add new node to the binary search tree

**public** **void** insert(**int** data)

{

          //Create a new node

          Node newNode = **new** Node(data);

          //Check whether tree is empty

**if**(root == **null**)

{

              root = newNode;

**return**;

            }

**else**

{

              //current node point to root of the tree

              Node current = root, parent = **null**;

**while**(**true**)

{

                  //parent keep track of the parent node of current node.

                  parent = current;

                  //If data is less than current's data, node will be inserted to the left of tree

**if**(data < current.data)

{

                      current = current.left;

**if**(current == **null**)

 {

                          parent.left = newNode;

**return**;

                      }

                  }

                  //If data is greater than current's data, node will be inserted to the right of tree

**else**

{

                      current = current.right;

**if**(current == **null**)

{

                          parent.right = newNode;

**return**;

                      }

                  }

              }

           }

      }

      //minNode() will find out the minimum node

**public** Node minNode(Node root)

{

**if** (root.left != **null**)

**return** minNode(root.left);

**else**

**return** root;

      }

      //deleteNode() will delete the given node from the binary search tree

**public** Node deleteNode(Node node, **int** value)

{

**if**(node == **null**)

{

**return** **null**;

           }

**else**

 {

              //value is less than node's data then, search the value in left subtree

**if**(value < node.data)

                  node.left = deleteNode(node.left, value);

              //value is greater than node's data then, search the value in right subtree

**else** **if**(value > node.data)

                  node.right = deleteNode(node.right, value);

              //If value is equal to node's data that is, we have found the node to be deleted

**else**

{

                  //If node to be deleted has no child then, set the node to null

**if**(node.left == **null** && node.right == **null**)

                      node = **null**;

                  //If node to be deleted has only one right child

**else** **if**(node.left == **null**)

 {

                      node = node.right;

                  }

                  //If node to be deleted has only one left child

**else** **if**(node.right == **null**)

 {

                      node = node.left;

                  }

                  //If node to be deleted has two children node

**else**

{

                      //then find the minimum node from right subtree

                      Node temp = minNode(node.right);

                      //Exchange the data between node and temp

                      node.data = temp.data;

                      //Delete the node duplicate node from right subtree

                      node.right = deleteNode(node.right, temp.data);

                  }

               }

**return** node;

          }

      }

      //inorder() will perform inorder traversal on binary search tree

**public** **void** inorderTraversal(Node node)

{

          //Check whether tree is empty

**if**(root == **null**){

              System.out.println("Tree is empty");

**return**;

           }

**else** {

**if**(node.left!= **null**)

                  inorderTraversal(node.left);

              System.out.print(node.data + " ");

**if**(node.right!= **null**)

                  inorderTraversal(node.right);

          }

      }

**public** **static** **void** main(String[] args)

{

          BinarySearchTree bt = **new** BinarySearchTree();

          //Add nodes to the binary tree

          bt.insert(50);

          bt.insert(30);

          bt.insert(70);

          bt.insert(60);

          bt.insert(10);

          bt.insert(90);

          System.out.println("Binary search tree after insertion:");

          //Displays the binary tree

          bt.inorderTraversal(bt.root);

          Node deletedNode = **null**;

          //Deletes node 90 which has no child

          deletedNode = bt.deleteNode(bt.root, 90);

          System.out.println("\nBinary search tree after deleting node 90:");

          bt.inorderTraversal(bt.root);

          //Deletes node 30 which has one child

          deletedNode = bt.deleteNode(bt.root, 30);

          System.out.println("\nBinary search tree after deleting node 30:");

          bt.inorderTraversal(bt.root);

          //Deletes node 50 which has two children

          deletedNode = bt.deleteNode(bt.root, 50);

          System.out.println("\nBinary search tree after deleting node 50:");

          bt.inorderTraversal(bt.root);

      }

}

**Output:**

Binary search tree after insertion:

10 30 50 60 70 90

Binary search tree after deleting node 90:

10 30 50 60 70

Binary search tree after deleting node 30:

10 50 60 70

Binary search tree after deleting node 50:

10 60 70

**Construct: Example problem**

Given an array of elements, we need to construct a BST.

**Given array:** 45, 10, 7, 90, 12, 50, 13, 39, 57

### **Binary Search Tree (BST) Traversal**

* Inorder Traversal
* Preorder Traversal
* PostOrder Traversal

All the above traversals use depth-first technique

The trees also use the breadth-first technique for traversal. The approach using this technique is called **“Level Order”** traversal.

#### **Inorder Traversal**

The inorder traversal approach traversed the BST in the order, **Left subtree=>RootNode=>Right subtree**. The inorder traversal provides a decreasing sequence of nodes of a BST.

**The algorithm InOrder (bstTree) for InOrder Traversal is given below.**

1. Traverse the left subtree using InOrder (left\_subtree)
2. Visit the root node.
3. Traverse the right subtree using InOrder (right\_subtree)

#### **Preorder Traversal**

In preorder traversal, the root is visited first followed by the left subtree and right subtree. Preorder traversal creates a copy of the tree. It can also be used in expression trees to obtain prefix expression.

**The algorithm for PreOrder (bst\_tree) traversal is given below:**

1. Visit the root node
2. Traverse the left subtree with PreOrder (left\_subtree).
3. Traverse the right subtree with PreOrder (right\_subtree).

#### **PostOrder Traversal**

The postOrder traversal traverses the BST in the order: **Left subtree->Right subtree->Root node**. PostOrder traversal is used to delete the tree or obtain the postfix expression in case of expression trees.

**The algorithm for postOrder (bst\_tree) traversal is as follows:**

1. Traverse the left subtree with postOrder (left\_subtree).
2. Traverse the right subtree with postOrder (right\_subtree).
3. Visit the root node